

Självständigt arbete på grundnivå

Applied Computer Engineering

Datateknik

Software engineering

IaC Template Suite for Credential-Free Workload Identity Provisioning

Albin Rönkvist & Piran Amedi



Mittuniversitetet

MID SWEDEN UNIVERSITY

MITTUNIVERSITETET

DSV Östersund

Examinator	Sergio Rico	sergio.rico@miun.se
Supervisor	Sergio Rico	sergio.rico@miun.se
Author 1	Albin Rönnkvist	alrn1700@student.miun.se
Author 2	Piran Amedi	roam2200@student.miun.se
Programme	Software Engineering, 180hp	
Course	DT002G, Applied Computer Engineering	
Field of study	Computer Engineering	
semester, year	VT, 2025	

Abstract

This report outlines the development of an Infrastructure as Code (IaC) template suite aimed at automating the provisioning of secure, credential-free workload identity and access management (IAM) within Microsoft Entra ID and Azure. Implemented with C# .NET and Pulumi IaC, the solution addresses the inefficiencies of manual infrastructure configuration, mitigates the security vulnerabilities associated with long-lived static credentials, and improves adaptability—ultimately reducing the time, effort, and financial overhead required to adapt and scale IAM systems. By incorporating IaC principles, Managed Identities (MIs) for internal workloads and Federated Identity Credentials (FICs) for external workloads, the template suite promotes a secure, consistent, and maintainable approach to workload IAM. The report provides a structured overview of the project's objectives, design rationale, implementation strategy, and evaluation methodology, emphasizing alignment with the defined Software Requirements Specification (SRS). The project successfully implemented a scalable IaC-based workload IAM template suite, simplifying adoption while eliminating reliance on manual configurations and long-lived static credentials.

Keywords: IAM, IaC, Credential-Free, Managed Identity, Federated Identity Credential, RBAC, Entra ID, Azure, Pulumi, C#, .NET

Table of Contents

1 Introduction	8
1.1 Problem Statement	8
1.2 Background and Motivation	8
1.3 Aim	9
1.4 Scope	9
1.5 Completeness Criteria	9
1.6 Time Plan	10
1.7 Outline	10
2 Methodology	11
2.1 Requirements	11
2.2 Development Method and Workflow	12
2.3 Tools and Architecture	13
3 Implementation	14
3.1 Design	14
3.1.1 Template suite & generating projects with custom parameters (FR1 & FR2)	14
3.1.2 Platform projects (FR3 & FR5)	14
3.1.3 Initialize and manage IAM Projects (FR4, FR5 & FR6)	16
3.1.3.1 lamEntrald Project	18
3.1.3.2 lamAzure Project	19
3.1.4 Putting it all together	21
3.2 Testing	22
3.2.1 Unit tests	22
3.2.2 Manual Testing in Real-World Scenarios (FR7 & FR8)	22
4 Results	24
5 Analysis and Discussion	26
References	27
Appendix A - Pulumì Issues & Workarounds	28
Appendix B - Azure Issues & Workarounds	29

Table of Tables

Table I	Functional Requirements
Table II	Non-Functional Requirements
Table III	Functional Requirements Result
Table IV	Non-Functional Requirements Result

Table of Figures

Figure 1	Gantt-diagram
Figure 2	Workflow
Figure 3	Template suite
Figure 4	Platform projects
Figure 5	IAM projects
Figure 6	IamEntrald project
Figure 7	IamAzure project
Figure 8	Least privilege access overview
Figure 9	Manual testing containerization infrastructure

Terminology

IAM	Identity and Access Management
IaC	Infrastructure as Code
RBAC	Role-Based Access Control
MI	Managed Identity
FIC	Federated Identity Credential
CI/CD	Continuous Integration and Continuous Delivery
CLI	Command-Line Interface

1 Introduction

1.1 Problem Statement

Identity and Access Management (IAM)¹ for workloads in cloud environments at scale presents significant challenges, especially when relying on manual configurations and long-lived credentials. While these traditional IAM approaches offer advantages—such as adoptability, flexibility, and full feature support—they struggle to scale efficiently in terms of security, consistency and maintainability. Adopting Infrastructure as Code (IaC)² principles and credential-free³ authentication can directly address these challenges. However, implementing these solutions can be complex, time-consuming, and costly, making it difficult for organizations to transition away from traditional IAM methods. This creates a clear need for a workload IAM solution that maintains ease of adoption while ensuring long-term scalability.

1.2 Background and Motivation

As organizations increasingly transition to cloud and hybrid environments, IAM for workloads—including applications, services, and automated processes—has become a critical component of securing cloud resources [1, 2]. Each workload must securely authenticate and operate with least privilege access to these resources. However, as systems scale and the number of workloads grows, managing IAM becomes increasingly complex. Two important factors that impact the complexity of workload IAM are the method used to provision identities and roles, and the method of which the identities authenticate. Traditionally, IAM infrastructure has been provisioned manually through cloud provider portals and ad hoc scripts. While this approach offers a quick setup with fully integrated features, it introduces significant challenges over time, including security risks, inconsistencies, and operational complexity [3, 4]. Manual modifications often lack peer reviews and oversight, making them error-prone and susceptible to misconfigurations and unauthorized changes, compromising security. Inconsistencies arise due to configuration drift, where settings change unpredictably, leading to snowflake environments that cannot be easily reproduced or automated. Without a standardized management process, tracking changes becomes challenging, increasing operational complexity and making auditing and compliance difficult. Integrating IaC into workload IAM directly addresses these challenges. By defining all IAM resources as code, organizations can establish a single source of truth that enables consistent and reproducible deployments across all environments. IaC further improves security and maintainability by providing enhanced visibility and standardized CI/CD workflows through peer-reviewed, version-controlled infrastructure code. [4, 5, 6]. Despite these benefits, adopting an IaC-driven IAM strategy involves some trade-offs. While IaC offers significant long-term benefits, it can be more difficult to adopt initially due to the substantial development effort, time investment, and required expertise [5, 6]. Furthermore, limited support for certain native cloud provider features may restrict functionality in specific scenarios.

Similar to the selection of provisioning methods, the choice of workload identity authentication mechanisms entails its own set of trade-offs and design considerations. Traditional workload IAM implementations rely on

¹ <https://learn.microsoft.com/entra/fundamentals/introduction-identity-access-management>

² <https://www.redhat.com/topics/automation/what-is-infrastructure-as-code-iac>

³ Credential-free authentication eliminates the need to manually manage secrets, certificates, and keys for secure service communication.

authentication using long-lived static credentials such as secrets and certificates [3, 4, 7]. While convenient, these credentials pose a significant security risk, as they require secure storage, regular rotation, and continuous manual management to prevent exposure. As organizations scale and the number of workloads increases, these challenges intensify, resulting in a weakened security posture, inconsistencies, and greater operational complexity [7]. In contrast, credential-free authentication replaces static credentials with ephemeral access⁴ methods, which rely on the identity provider or platform to automatically issue and rotate short-lived credentials. This model applies to both internal workloads—such as cloud-managed identities⁵—and external workloads via workload identity federation⁶, thereby eliminating the need for manual credential storage and rotation. As a result, organizations can eliminate the risk of credential exposure and minimize operational overhead as systems scale [8]. However, credential-free authentication also introduces certain challenges. Its implementation often requires more complex identity provider-specific configurations and may involve managing a larger set of identities compared to traditional approaches. Moreover, limited feature support across diverse workloads can constrain flexibility in some environments. To address the limitations of both traditional and modern approaches, there is a clear need for a secure, consistent, and maintainable workload IAM solution—one that supports scalability while preserving the ease of adoption found in conventional methods.

1.3 Aim

This project will develop an IaC-driven template suite that automates the setup of credential-free workload IAM infrastructure. Ultimately, the goal is to provide organizations with a solution that utilizes modern IaC and credential-free principles for improved scalability while retaining the ease of adoption of traditional approaches, reducing development time and costs.

1.4 Scope

The scope of this project includes developing a fully functional suite of IaC templates to provision workload identities and roles in Microsoft Entra ID and Azure. It comprises a set of integrated templates, each designed for distinct but complementary functionalities within workload IAM. This template suite integrates closely with C# .NET, Pulumi, GitLab, and Docker. The scope excludes non-Azure cloud environments, and other providers and tools. Additionally, the project is confined to the provisioning phase of IAM and does not encompass post-deployment monitoring or security auditing activities.

1.5 Completeness Criteria

The project will be considered complete when a fully functional suite of Pulumi- and C# .NET-based IaC templates is available for installation, allowing organizations to set up provisioning of credential-free workload identities⁷ with least privilege⁸ Role-Based Access Control (RBAC)⁹ for accessing Azure resources. The solution must fulfill the following requirements:

⁴ <https://www.entitle.io/resources/glossary/ephemeral-access>

⁵ <https://learn.microsoft.com/en-us/entra/identity/managed-identities-azure-resources/overview>

⁶ <https://learn.microsoft.com/en-us/entra/workload-id/workload-identity-federation>

⁷ <https://learn.microsoft.com/entra/workload-id/workload-identities-overview>

⁸ <https://learn.microsoft.com/entra/identity-platform/secure-least-privileged-access>

⁹ <https://learn.microsoft.com/azure/role-based-access-control/overview>

- 10

2 Methodology

This section provides a detailed breakdown of the requirements, development methodology and workflows, and tooling and architecture that shaped the project's implementation.

2.1 Requirements

To define a precise specification of the product, the following functional and non-functional requirements were formulated (see Tables I and II).

TABLE I
FUNCTIONAL REQUIREMENTS

ID	TITLE	DESCRIPTION
FR1	Install Template Suite	Install the template suite as a single package using the .NET template engine.
FR2	Generate Projects with Custom Parameters	Create projects from templates with configurable parameters.
FR3	Distribute Reusable Components	Publish reusable components as NuGet packages.
FR4	Initialize IAM Projects	Run initializer programs to self-assign identities and access via Global Administrator, enabling future contributors to use the assigned FIC.
FR5	Include CI/CD Pipelines	Ensure all generated projects have integrated CI/CD pipelines for automated build, test, and deployment.
FR6	Contribute to IAM Projects	IAM Administrators should be able to provision necessary Microsoft Entra ID and Azure resources from IAM projects.
FR7	Provision Microservice Resources	Enable operations developers to provision microservice resources with assigned identities and roles via CI/CD automation.
FR8	Access Microservice Resources	Developers should securely access Azure resources within internal workloads using Managed Identities (MIs).

TABLE II
NON-FUNCTIONAL REQUIREMENTS

ID	TITLE	DESCRIPTION
QR1	Optimized Performance	Ensure infrastructure deployment and CI/CD execution complete within 5 minutes.
QR2	Code Readability	Maintain clear, well-documented, and structured code for ease of understanding and extension.
QR3	Modular & Reusable Components	Maximize reuse of resource builders, naming conventions, and other components.
QR4	Testability	Maintain a test environment with configurations consistent with production.
QR5	Comprehensive Documentation	Provide detailed setup, usage, and troubleshooting guides.
QR6	Credential-free Workload Identities	Use credential-free workload identities throughout the entire infrastructure, for Azure resource access in both internal and external workloads.
QR7	Least Privilege Access	Enforce least privilege access when provisioning identities and roles.

2.2 Development Method and Workflow

The development process adopted DevOps practices and tools to manage the project lifecycle across its Plan, Develop, Deliver, and Operate phases, as illustrated in Fig. 2. To effectively plan development tasks, the team used the agile framework Kanban, organizing work into epics and progressing through well-defined stages: “TODO”, “IN PROGRESS”, “READY FOR REVIEW”, “READY FOR RELEASE”, and “DONE”. Each task was assigned to a specific developer and prefixed with the repository name to clarify its project scope. For developing and delivering the planned tasks, the project adopted a CI/CD approach with Trunk-Based Development (TBD) as its version control strategy. Developers worked on short-lived branches, regularly rebasing onto the latest development or main (trunk) branch to keep their code up to date and reduce merge commits and conflicts. Immediately upon branch creation, a draft merge request (MR) was created, prefixed

with the corresponding task ID, which integrated the CI/CD tool with the Kanban tool for improved traceability. Additionally, a CI pipeline was automatically triggered on each MR update, validating changes by running builds and tests. Before merging into the trunk, every MR underwent a structured peer review process, requiring at least one reviewer's approval. To enhance real-time collaboration, automated notifications were sent via communication channels whenever a task moved to "READY FOR REVIEW", ensuring that reviews were conducted promptly. Once an MR was approved, the assignee informed the team via a dedicated release queue communication channel to prevent deployment conflicts before merging it into the trunk, which then triggered an automated CI/CD pipeline to deploy the changes. By integrating Kanban, CI/CD, Trunk-Based Development, and automated communication, the team maintained an efficient, scalable, and collaborative DevOps environment, enabling rapid iteration, high-quality code, and continuous software evolution.

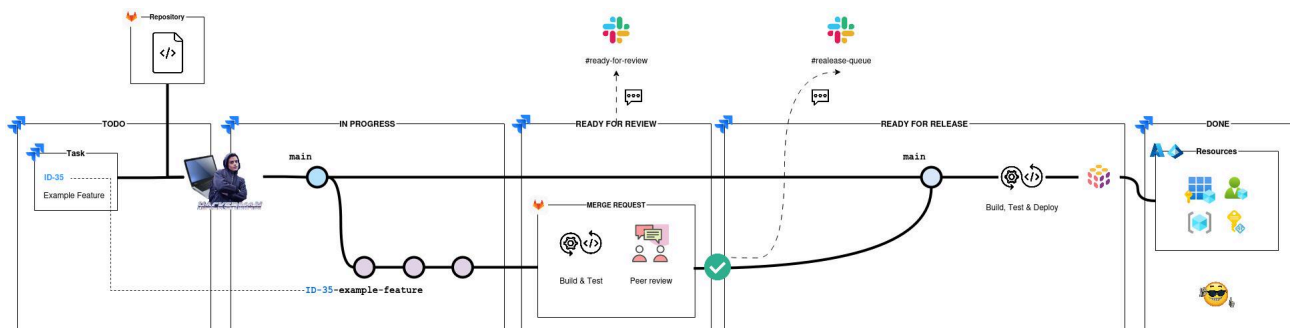


Fig. 2. Workflow

2.3 Tools and Architecture

The development environment for this project was designed to support a DevOps-driven workflow, integrating tools that streamlined the entire DevOps lifecycle. Jira was used for task management, while Slack facilitated team communication, providing automated updates for workflow events, such as when tasks moved to "READY FOR REVIEW". The team primarily used JetBrains Rider as the integrated development environment (IDE) for writing code. Git was utilized for version control, with GitLab serving as the DevOps platform, managing source code, merge requests, code reviews, CI/CD pipelines, and package registries. The project was developed using C# .NET 9.0, deeply integrated with Pulumi, Microsoft Entra ID, Microsoft Azure, and Docker. To enhance development efficiency and avoid reinventing the wheel, the codebase incorporated various well-established .NET libraries, including: GitVersion for semantic versioning, xUnit as the testing framework, with NSubstitute for mocking and Shouldly for fluent assertions, Ardalis.SmartEnum for smart enums, FluentValidation for input validation, and more.

The template suite was designed with a microservices architecture in mind, ensuring that each microservice had a dedicated IAM configuration tailored to its specific responsibility. Dependencies between microservices were managed through distributed NuGet packages, which followed semantic versioning and were hosted in a private GitLab package registry. Additionally, the template suite adopted a micro-stack architecture¹², a Pulumi approach similar to microservices but applied at the Pulumi stack level, breaking the suite into smaller, independently managed components. To enable communication between micro-stacks, stack

¹² <https://www.pulumi.com/docs/iaac/using-pulumi/organizing-projects-stacks/#micro-stacks>

references¹³ were utilized and documented as contract NuGet packages, eliminating reliance on magic strings. Lastly, the project heavily utilized the builder pattern to define common resources with flexibility.

3 Implementation

This section outlines the design and testing of the system, detailing how it was implemented to fulfill both functional and non-functional requirements. It includes infrastructure design, IAM provisioning, and CI/CD automation, along with validation through unit tests and manual tests in a real-world scenario.

3.1 Design

This section outlines the implementation of the IAM template suite. It begins by examining the organization and packaging of the templates, as well as their ability to generate projects with custom parameters. Next, it explores the template contents, starting with Platform NuGet packages for reusable components, including automated workflows through CI/CD pipelines. The focus then shifts to IAM projects, detailing their initialization process, contribution methods, and the integration of automation via CI/CD pipelines. Finally, the section concludes with an architectural summary, providing a comprehensive overview of how these components work together to achieve least privilege access.

3.1.1 Template suite & generating projects with custom parameters (FR1 & FR2)

As shown in Fig. 3, the IAM template suite functions as a container for multiple integrated templates, designed to simplify the creation of projects with custom configurations.

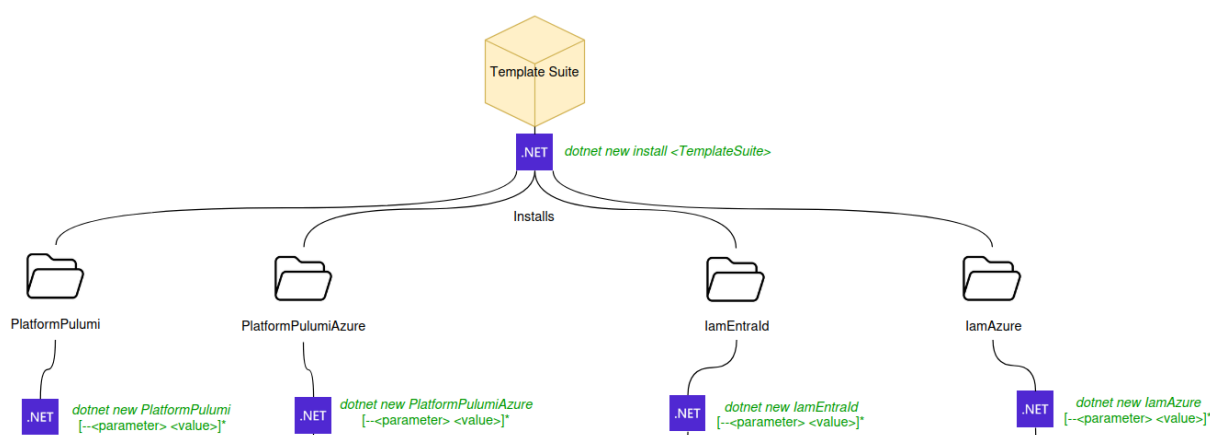


Fig. 3. Template suite

The suite utilizes the .NET Template Engine and is distributed as a template package via NuGet, making it easily installable using the `dotnet new` command. The suite includes four templates categorized into two sections: *platform* projects and *IAM* projects, which will be discussed in more detail in the following sections of this report. Each template can be generated using the `dotnet new` command with customizable parameters such as organization name, private NuGet feed source, and other configurable values.

¹³ <https://www.pulumi.com/tutorials/building-with-pulumi/stack-references/>

3.1.2 Platform projects (FR3 & FR5)

The platform projects, **PlatformPulumi** and **PlatformPulumiAzure**, are reusable packages designed to standardize conventions and components across the IAM architecture in the template suite. **PlatformPulumi** serves as the base package for using Pulumi, incorporating naming conventions and essential Pulumi-specific components. **PlatformPulumiAzure** extends this with Azure-specific configurations, ensuring seamless integration with Microsoft's cloud platform. As illustrated in Fig. 4, both projects are distributed as NuGet packages, leveraging semantic versioning with GitVersion¹⁴, where major and minor versions are manually controlled via git tags, while patch versions are automatically incremented with each build. Integrated GitLab CI/CD pipelines handle builds, testing, and deployments to a private GitLab package registry, with pre-releases manually triggered in merge requests and releases automatically triggered upon merging to the main branch.

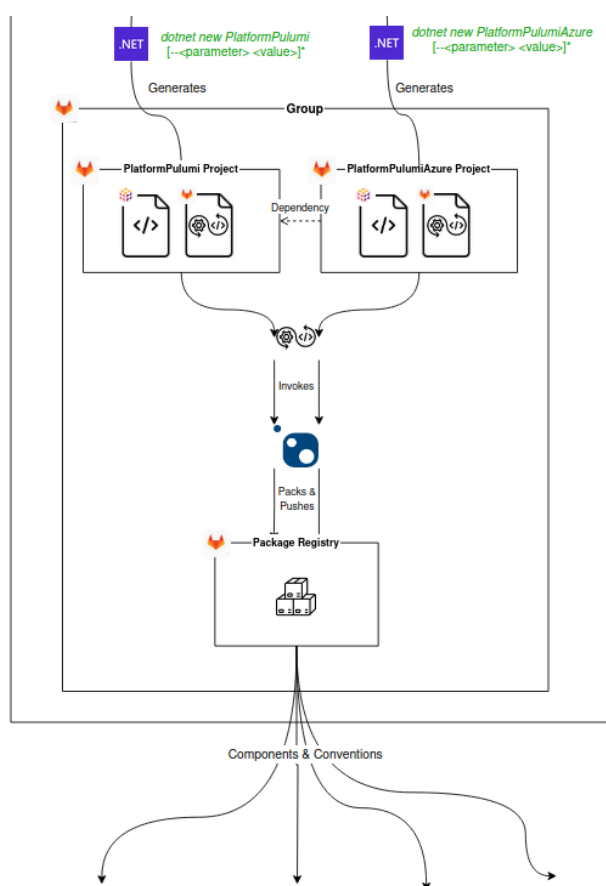


Fig. 4. Platform projects

¹⁴ <https://gitversion.net/docs/usage/cli/installation>

Packages are published to project-level¹⁵ NuGet endpoints in GitLab, which are required for uploading to the registry. Once published, these packages become automatically available through the group-level¹⁶ NuGet endpoint, enabling consumption across multiple projects using a single endpoint. To support this, each consuming project's NuGet.Config is updated to include the group-level source. Additionally, GitLab's CI job token¹⁷ is granted access by adding the group to each project's CI/CD job token allowlist¹⁸, allowing pipelines to retrieve group-level packages during builds. This approach is used across the entire project.

3.1.3 Initialize and manage IAM Projects (FR4, FR5 & FR6)

The IAM projects, **iamEntraId** and **iamAzure**, provide an IaC-approach for managing credential-free workload identities and access in Microsoft Entra ID and Azure, as illustrated in Fig. 5.

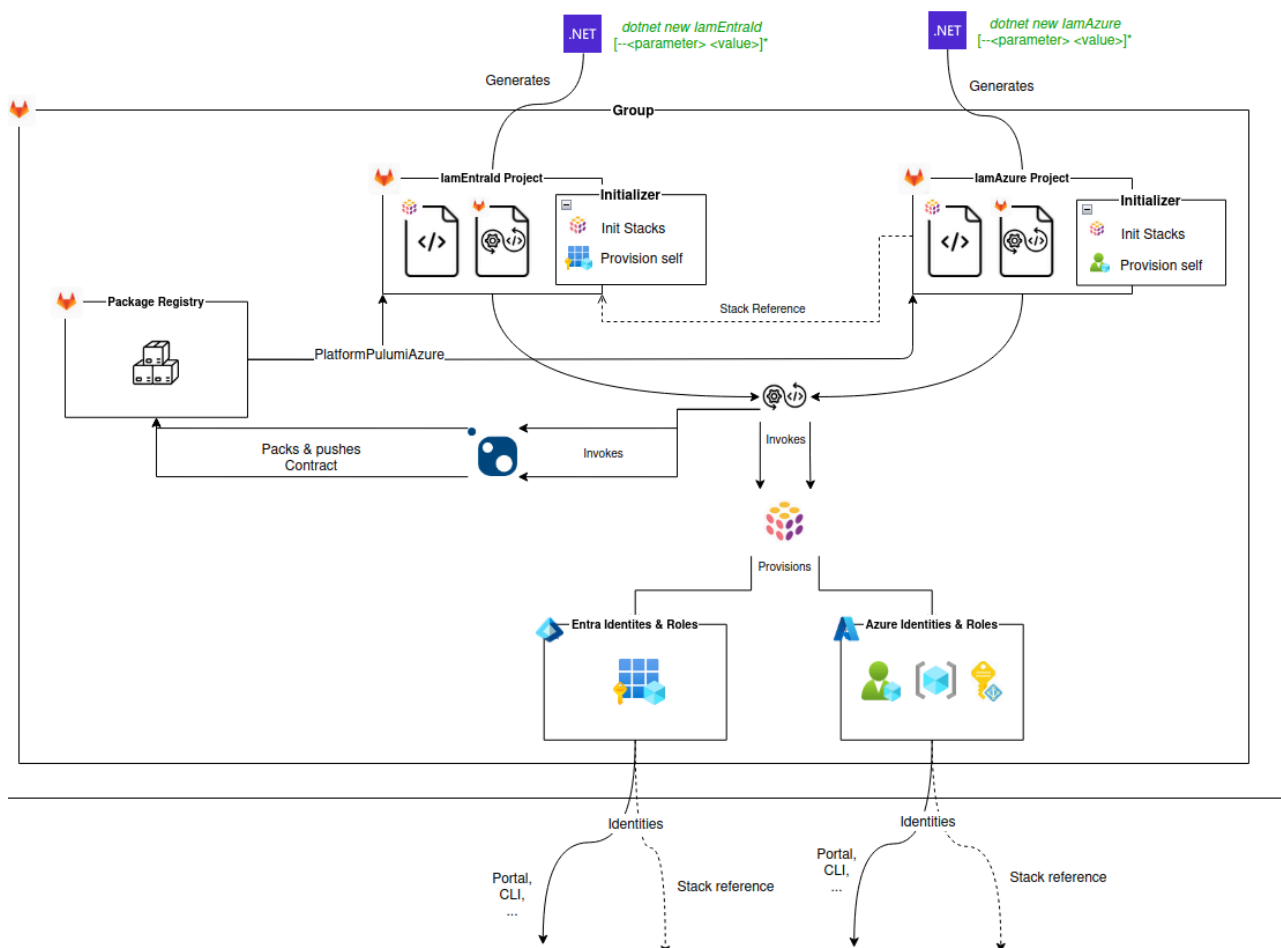


Fig. 5. IAM projects

¹⁵ https://docs.gitlab.com/user/packages/nuget_repository/#project-level-endpoint

¹⁶ https://docs.gitlab.com/user/packages/nuget_repository/#group-level-endpoint

¹⁷ https://docs.gitlab.com/ci/jobs/ci_job_token/

¹⁸ https://docs.gitlab.com/ci/jobs/ci_job_token/#add-a-group-or-project-to-the-job-token-allowlist

These projects automate the provisioning of necessary resources within their respective domains, while adhering to the same standardized structure, with internal projects handling distinct responsibilities:

- **Pulumi** – Defines and provisions resources, serving as the source of truth.
- **Pulumi.Contract** – Defines contracts for stack references and outputs produced by the **Pulumi** project.
- **Pulumi.Initializer** – Initializes stacks and Pulumi-related configurations, and enables self-assigning identities and roles.

The main **Pulumi** projects are executed using `Pulumi.Deployment.RunAsync<MainStack>()`, which serves as the standard entry point for Pulumi .NET applications. In this setup, the `MainStack` class functions as the root of the infrastructure definition, grouping all resources under a unified deployment scope. To enhance maintainability and structure, resources are organized into component resources¹⁹. This design establishes clear parent-child relationships and logical groupings, allowing related infrastructure elements to be encapsulated within modular units. Within each component, custom resources²⁰ managed by Azure's `azure-native`²¹ and `azuread`²² providers, define the underlying cloud infrastructure. Additionally, these resources generate output values that can be consumed as inputs by other resources, and are explicitly exposed through the component's `RegisterOutputs()` method. To further abstract complexity and promote reusability, the builder pattern is applied throughout the codebase. This pattern encapsulates resource construction while maintaining flexibility through fluent `With()` methods, enabling a clean, declarative style.

With resources defined in the **Pulumi** projects, necessary values are exposed as stack outputs from the `MainStack` class using the `[Output]` attribute. The overall system follows a micro-stack architecture, where independently managed Pulumi stacks communicate through stack references and stack outputs, ensuring modularity and separation of concerns. However, because Pulumi stack outputs in .NET do not support strongly typed values, they must be retrieved as dictionaries or primitive types and then manually parsed or deserialized. To enforce stricter contracts and ensure structured outputs, the **Pulumi.Contract** projects defines stack references and outputs, distributing them as a NuGet package for use in other projects. This approach avoids reliance on magic strings and maintains consistency across integrations.

Although the **Pulumi** and **Pulumi.Contract** projects contain all the functional logic for provisioning resources, the **Pulumi** projects depend on an identity with sufficient permissions to deploy the defined infrastructure. However, without pre-existing permissions, these projects cannot assign identities or roles to themselves. To address this limitation, the **Pulumi.Initializer** plays a critical role in enabling self-provisioning. A *Global Administrator*—typically the default privileged account in a new Microsoft Entra ID tenant—is required to execute the initializer. During this process, the initializer creates a Pulumi stack, which serves as the deployment environment, and runs the **Pulumi** project with elevated privileges. This allows it to provision the necessary IAM resources, including self-assigning identity and roles used for future deployments. Once the provisioning is complete, the initializer updates the project's configuration files (`Pulumi*.yaml`) with the appropriate settings and outputs a Client ID for the self-assigned workload identity. After this one-time setup,

¹⁹ <https://www.pulumi.com/docs/iac/concepts/resources/components/>

²⁰ <https://www.pulumi.com/docs/iac/concepts/resources/>

²¹ <https://www.pulumi.com/registry/packages/azure-native/api-docs/provider/>

²² <https://www.pulumi.com/registry/packages/azuread/api-docs/provider/>

subsequent deployments can proceed automatically through a CI/CD pipeline configured with the self-assigned workload identity, without requiring further intervention from a Global Administrator.

With all manual development and initial provisioning completed, the **Pulumi** and **Pulumi.Contract** projects are integrated into a CI/CD pipeline, automating building, testing, previewing, and deployment. The pipeline builds, tests and previews (`pulumi preview`) the **Pulumi** project during an MR to verify changes before applying them. Once merged into the main branch, the deployment process is automatically triggered (`pulumi up`). Resource provisioning access is configured by manually adding the self-assigned workload identity's Client ID—generated by the initializer—to the CI/CD pipeline configuration. Pulumi access in the CI/CD pipeline is configured using a Personal Access Token (PAT), selected as a cost-effective solution in lieu of enterprise authentication options—this decision is further explained later in the report. The PAT is stored as a masked and hidden CI/CD variable²³ named `PULUMI_ACCESS_TOKEN` at the GitLab group level. Once saved, the token is automatically hidden from job logs and cannot be viewed or retrieved through the GitLab interface, ensuring it remains protected during pipeline execution. The **Pulumi.Contract** NuGet packages follow the same deployment workflow as the Platform projects; however, they are only published when a change is detected within the specific project. This is controlled using the changes rule²⁴ with the path: `Pulumi.Contract/**/*`.

3.1.3.1 lamEntrald Project

The **lamEntrald** project is responsible for identity and access management (IAM) within Entra ID, as shown in Fig. 6. It provisions workload identities that use credential-free federated authentication while assigning necessary roles and permissions.

²³ <https://docs.gitlab.com/ci/variables/#hide-a-cicd-variable>

²⁴ <https://docs.gitlab.com/ci/yaml/#ruleschanges>

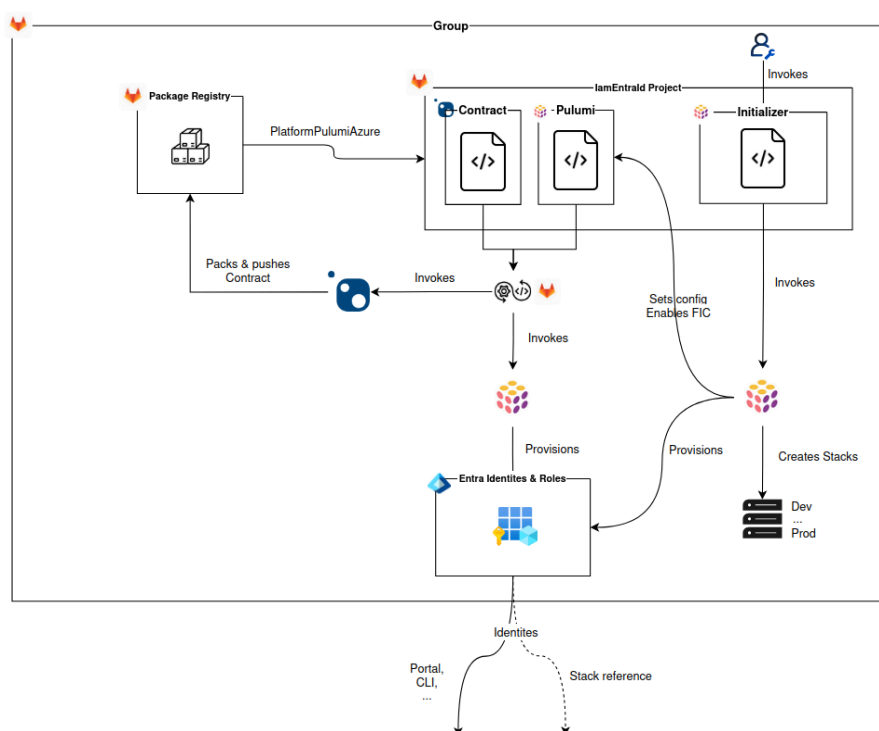


Fig. 6. IamEntrald project

The **Pulumi** project defines the resources, including applications, service principals, FICs, directory roles, and API permissions. The project also includes users and groups resources, although this is outside the scope of the project and will not be discussed further. The application represents the workload identity globally across all tenants, while the service principal provides a local representation within a specific tenant. Federated authentication is enabled for the application by creating a flexible FIC,²⁵ establishing a trusted relationship between Entra ID and the GitLab IdP, using a claims matching expression to specify authentication conditions. For example, to restrict access to a specific group and project across any branch, the expression is defined as: `project_path:{group}/{project}:ref_type:branch:ref:*`. Additionally, the **IamEntrald** service principal is assigned directory roles and application API permissions, providing it with administrative privileges and the ability to perform necessary API actions without requiring a signed-in user. Moreover, it automatically grants admin consent, reducing the need for manual intervention. As outlined in earlier sections, the **Pulumi** project cannot assign roles to itself without existing permissions. To address this, a *Global Administrator* must initially execute the **Pulumi.Initializer**, which provisions the workload identity along with the required directory roles, application API permissions, and admin consent. Once this setup is complete, subsequent deployments can run automatically through the CI/CD pipeline without requiring further involvement from a *Global Administrator*. Once the **Pulumi** project is initialized and all required IAM resources are provisioned, it exposes stack outputs—most notably, the application identities. These outputs enable seamless integration with other projects by providing the necessary values, such as the Application Client ID, Service Principal Object ID, and the underlying FIC, which are used to authenticate

to Azure.

3.1.3.2 lamAzure Project

The **lamAzure** project is responsible for managing workload identities and access control within Azure, complementing the **lamEntrald** project by provisioning the necessary Azure-specific resources, as shown in Fig. 7. It automates the creation of Resource Providers, Resource Groups, Managed Identities, and Azure Role-Based Access Control (RBAC) assignments, ensuring that workload identities have the required permissions to operate securely within an Azure subscription.

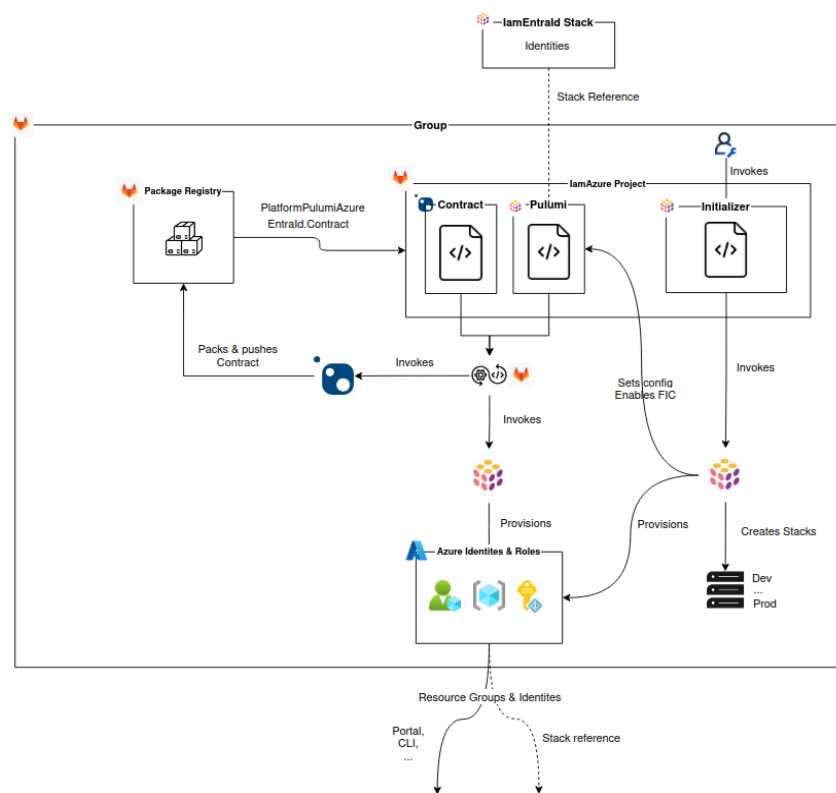


Fig. 7. lamAzure project

The **Pulumi** project provisions the necessary Azure Resource Providers to enable the deployment of services such as compute, networking, and storage. To ensure organizational clarity and enforce scoped access control, it also creates Resource Groups, which serve as logical containers for grouping related resources. Additionally, the project sets up user-assigned managed identities—standalone Azure identities that can be explicitly linked to multiple services. These identities support credential-free authentication by allowing Azure resources, such as container apps, functions, or virtual machines, to securely access other services without relying on secrets or certificates. To enforce security best practices and the principle of least privilege, the project assigns Azure RBAC roles to each identity according to its defined scope of responsibility.

For the **lamAzure.Pulumi** project to function independently, it retrieves its workload identity from the **lamEntraId.Pulumi** project via stack reference, where FIC is enabled and scoped to its corresponding GitLab repository. This identity is then self-assigned the *Subscription Owner* role in the infrastructure code, allowing it full administrative control over the designated subscription. However, since an identity cannot assign itself a role it does not yet possess, the project must first execute **Pulumi.Initializer**. This initializer process is triggered by the *Global Administrator*, which grants the **lamAzure** identity the necessary permissions to manage resources and roles within the subscription. Once initialized, future deployments can proceed automatically through the CI/CD pipeline, authenticating with the assigned workload identity. Finally, the project produces stack outputs in the form of provisioned resource groups and managed identities.

3.1.4 Putting it all together

Bringing everything together, the platform projects provide the foundational conventions and reusable components that the IAM projects and other projects rely on. The IAM projects are at the core of this model, each managing identities and access within their specific domains. **lamEntraId** governs identity and access within Microsoft Entra ID. It ensures that only Entra ID resources and roles are managed under its scope, preventing unnecessary access to Azure resources. Meanwhile, **lamAzure** operates within the Azure ecosystem. As a *Subscription Owner*, it has full control over Azure resources and roles, but its scope remains strictly limited to Azure and the specified subscriptions, with no direct influence over Entra ID or other subscriptions in the tenant. Other workloads—such as application and infrastructure projects—consume these IAM resources while following the least privilege model. Instead of obtaining unrestricted access at the subscription level, each workload is assigned ownership over a specific Resource Group (RG). These workloads can only create and manage resources and roles within their assigned RG, ensuring granular control and preventing privilege escalation. Additionally, they are restricted to pre-approved resource providers, meaning they can only deploy services that are defined in the **lamAzure** project. This structured approach enables seamless collaboration by allowing teams to operate independently within clearly defined boundaries, as illustrated in Fig. 8. The platform projects provide the guardrails, the IAM projects enforce access control, and other various projects consume these resources securely, creating a least privilege architecture that scales efficiently across Microsoft Entra ID and Azure.

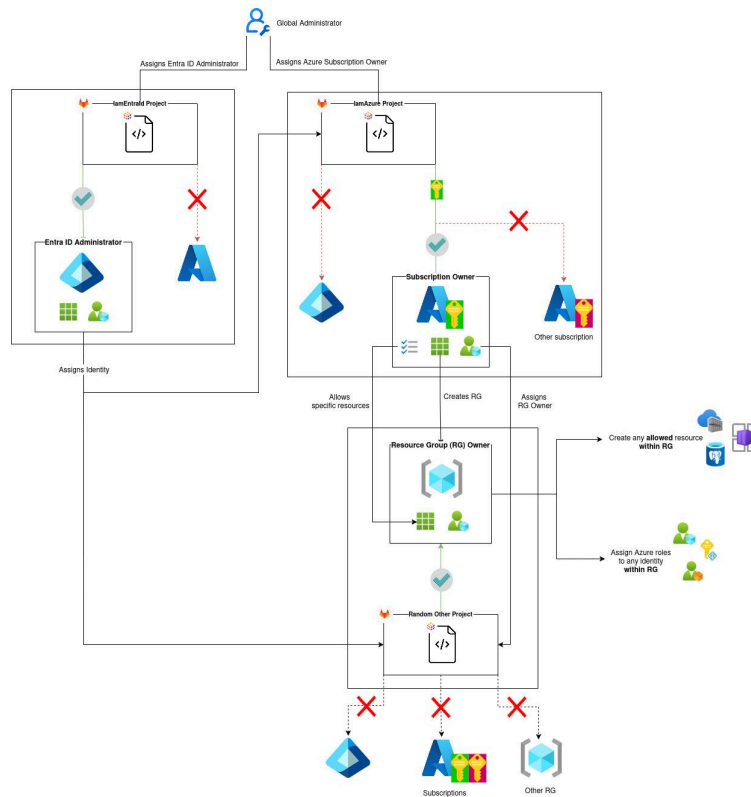


Fig. 8. Least privilege access overview

3.2 Testing

The template suite underwent rigorous testing through unit tests and manual validation. Unit tests ensured infrastructure correctness using mocked, in-memory execution, while manual tests validated real-world functionality in production-like environments, confirming that identities and access work as expected across Entra ID and Azure.

3.2.1 Unit tests

To ensure correctness, the IAM projects include respective internal **Pulumi.Test** projects, providing unit tests for validating Pulumi infrastructure definitions. These tests run entirely in memory, replacing external dependencies with mocks. Since the infrastructure code is written in C#, the unit tests leverage existing tools, such as the xUnit testing framework with NSubstitute for mocking and Shouldly for fluent assertions.

3.2.2 Manual Testing in Real-World Scenarios (FR7 & FR8)

To validate real-world usability, manual validation tests were conducted in test environments that mirrored production configurations. To achieve this, a containerization infrastructure was provisioned, as illustrated in Fig. 9.

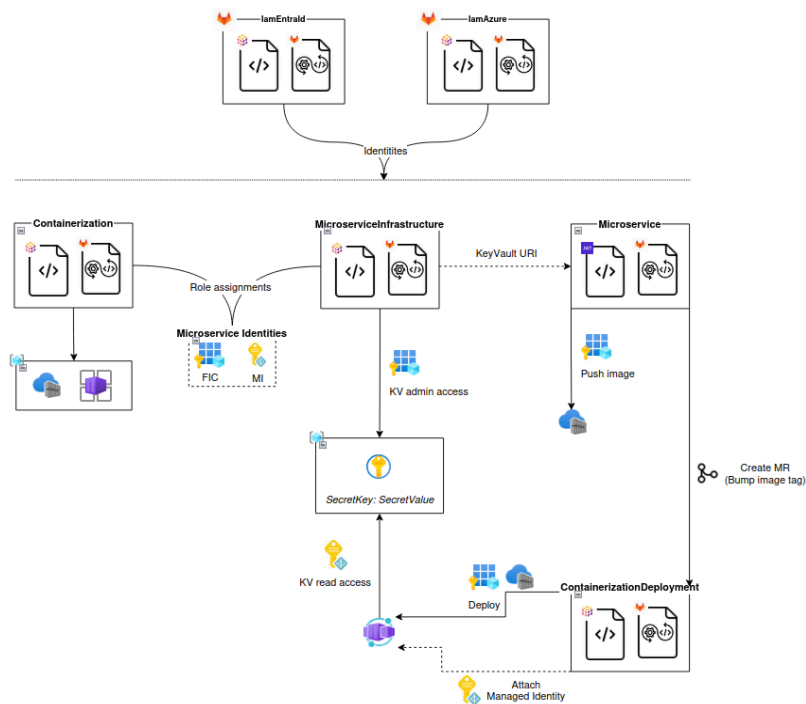


Fig. 9. Manual testing containerization infrastructure

This infrastructure consisted of four projects:

- **Containerization** – An IaC project responsible for provisioning container-related resources.
- **Microservice** – A containerized ASP.NET Core Web API.
- **MicroserviceInfrastructure** – An IaC project responsible for provisioning infrastructure for the microservice.
- **ContainerizationDeployment** – An IaC project responsible for deploying the containerized microservice.

The **Containerization** project served as the foundation of this infrastructure, leveraging identities and roles provisioned by the IAM projects to create an Azure Container Registry (ACR) and an Azure Container Apps Environment (CAE). These resources were assigned the necessary roles and permissions, allowing other projects to securely publish Docker images and deploy them as containerized applications. Following this, a microservice and its corresponding infrastructure were provisioned. Using Pulumi with FIC authentication and assigned roles, the **MicroserviceInfrastructure** project created an Azure Key Vault, storing a secret to be accessed by the microservice. The **Microservice** itself was a simple ASP.NET Core Web API, configured to retrieve secrets from the Key Vault using a managed identity, eliminating the need for direct credential storage. Once the microservice was containerized, its CI/CD pipeline pushed the resulting image to the Azure Container Registry (ACR) and triggered a merge request to update the image tag in the **ContainerizationDeployment** project. Since **ContainerizationDeployment** defined the container app using Pulumi IaC, it seamlessly deployed the updated containerized microservice to the CAE through the CI/CD pipeline. With the entire system in place, the test successfully confirmed end-to-end functionality. The running microservice container was able to retrieve the Key Vault secret, validating that the infrastructure, identity management, and access controls were functioning as intended in a real-world deployment scenario.

4 Results

The implementation of the system successfully met all functional requirements and most non-functional requirements (with room for interpretation), resulting in a scalable workload IAM template suite. It ensured security, consistency, and maintainability by leveraging IaC principles, while further enhancing security and maintainability with credential-free workload authentication. Adoption was simplified through customizable templates, allowing teams to quickly generate projects aligned with IAM best practices. Additionally, necessary workarounds were implemented to support missing features, ensuring complete integration across Microsoft Entra ID and Azure.







The status and rationale behind achieving these functional and non-functional requirements are detailed in Table III and IV below.

TABLE III
FUNCTIONAL REQUIREMENTS RESULT

ID	STATUS	RATIONALE
FR1	✓	The IAM template suite was successfully implemented as a single installable package using the .NET template engine.
FR2	✓	Projects could be generated from templates with configurable parameters, allowing customization based on specific use cases.
FR3	✓	Reusable components were distributed as Platform NuGet packages.
FR4	✓	The IAM projects were designed to self-assign identities and access by executing initializer programs.
FR5	✓	Each generated project included an integrated CI/CD pipeline, automating build, test, and deployment processes.
FR6	✓	IAM administrators could provision and manage Microsoft Entra ID and Azure resources directly from IAM projects.
FR7	✓	Operations developers were able to provision microservice-specific resources in assigned Resource Groups, with predefined identities and roles.
FR8	✓	Developers were able to securely access Azure resources within internal workloads using Managed Identities (MIs).

TABLE IV
NON-FUNCTIONAL REQUIREMENTS RESULT

ID	STATUS	RATIONALE
QR1	✓	Infrastructure deployment and CI/CD execution times were optimized to complete within five minutes.

QR2		The system strived towards clear, well-documented, and structured code. However, no metrics were used to verify this.
QR3		A modular design maximized the reuse of resource builders and naming conventions, reducing duplication across projects.
QR4		A dedicated test environment was maintained, ensuring configuration consistency with future staging or production environments.
QR5		The system included detailed documentation covering setup, usage, and troubleshooting.
QR6		Credential-free workload identities were enforced across the entire IAM infrastructure, ensuring secure access without the use of static credentials.
QR7		IAM provisioning followed least privilege principles, aiming to only assign necessary roles and permissions to workload identities. However no risk detection tools were used to verify this.

5 Analysis and Discussion

The project successfully implemented a scalable IaC-based workload IAM template suite, simplifying adoption while eliminating reliance on manual configurations and long-lived static credentials. While the system effectively addresses the primary challenges, certain workarounds were necessary due to third-party provider feature support limitations, software bugs, and financial constraints. Throughout the project, Pulumi and Azure-related issues required deviations from the intended approach to ensure full functionality. Details on these challenges and workarounds can be found in [Appendix A](#) and [Appendix B](#). Additionally, some important security features were restricted to paid plans, preventing the use of more streamlined and secure solutions. A notable example is Pulumi's OIDC-based integration, which is limited to enterprise users, requiring the use of Pulumi PATs instead.

Several alternative solutions were explored but ultimately not pursued due to time constraints and project scope. While GitLab pipelines functioned as expected, there is room for optimization in readability and efficiency. Some scripts were lengthy and difficult to interpret, but given that the project's primary focus was on workload IAM, refining pipeline code was not prioritized. The system could also benefit from improved integration testing and expanded test coverage. However, each organization has its own testing standards, and adding excessive pre-built tests could become cumbersome, as users would likely need to remove components that do not align with their needs. Similarly, while expanding support for additional cloud providers, DevOps platforms, and .NET versions would increase versatility, it was beyond the scope of this project and would have required significantly more development time.

The challenges faced in this project, along with the alternatives explored, underscore the common complexities of managing cloud infrastructure. The need for workarounds due to missing third-party features and software bugs reflects a broader industry challenge—carefully selecting tools and conducting rigorous testing before deployment. Additionally, feature restrictions in free-tier services compel organizations to balance cost efficiency against security best practices, leading to trade-offs that could impact security. Lastly, designing templates requires striking the right balance—offering sufficient functionality without excessive complexity, ensuring adaptability and flexibility for diverse use cases.

Several areas could be explored to further enhance and expand this project. Increased automation could move the system toward a fully hands-off IAM solution, particularly by provisioning GitLab resources using Pulumi, eliminating remaining manual steps. Expanding compatibility with additional cloud providers, DevOps platforms, and tools would improve adaptability across different environments. Additionally, while user and group management for human identities was integrated, it was not the primary focus and could be further enhanced by strengthening security and privacy measures. Finally, implementing logging, monitoring, and auditing capabilities would enhance visibility into IAM operations, improving security tracking, compliance, and anomaly detection.

References

- [1] S. Ahmadi, "Systematic literature review on cloud computing security: Threats and mitigation strategies," *Journal of Information Security*, vol. 15, pp. 148–167, Mar. 2024.
- [2] A. Puchta, F. Böhm, and G. Pernul, "Contributing to current challenges in identity and access management with visual analytics," *Proc. 33rd IFIP WG 11.3 Conf. Data Appl. Secur. Privacy (DBSec 2019)*, vol. 11559, pp. 221–239, Cham, Switzerland, Jul. 2019.
- [3] H. Wang, B. Kishiyama, D. Lopez, and J. Yang, "An overview of Infrastructure as Code (IaC) with performance and availability assessment on Google Cloud Platform," *Proc. 2nd Int. Conf. Adv. Comput. Res. (ACR'24)*, pp. 497–514, Mar. 2024.
- [4] D. Verner, "The development of Infrastructure as Code practices for improving IT infrastructure management efficiency," *Norwegian J. Dev. Int. Sci.*, no. 142, p. 75–78, 2024.
- [5] M. R. Hasan and M. S. Ansary, "Cloud infrastructure automation through IaC (infrastructure as code)," *Int. J. Comput.(IJC)*, vol. 46.1, pp. 34–40, 2023.
- [6] D. Ljunggren, 'DevOps: Assessing the Factors Influencing the Adoption of Infrastructure as Code, and the Selection of Infrastructure as Code Tools : A Case Study with Atlas Copco', Dissertation, 2023.
- [7] J. Lepiller, R. Piskac, M. Schäf, and M. Santolucito, "Analyzing Infrastructure as Code to prevent intra-update sniping vulnerabilities," *Proc. 27th Int. Conf. Tools Algorithms Constr. Anal. Syst. (TACAS 2021)*, pp. 105–123, Mar.–Apr. 2021. Springer.
- [8] P. Somasundaram, "Unified Secret Management Across Cloud Platforms: A Strategy for Secure Credential Storage and Access," *Int. J. Comput. Eng. Technol*, vol. 15, pp. 5–12, 2024.

Appendix A - Pulumi Issues & Workarounds

Description	Workaround	Further read
<p>Pulumi's Azure Native provider currently lacks an equivalent to <code>core.ResourceProviderRegistration</code>, which was available in the classic Azure provider.</p> <p>The native provider only supports PUT-based resources and does not handle POST endpoints, which are required for resource provider registration.</p>	<p>Either manually register resource providers or resort to using Azure CLI or other alternatives.</p> <p>While this workaround ensures successful role provisioning, it increases complexity and makes it difficult to detect configuration drift.</p>	<p>https://github.com/pulumi/pulumi-azure-native/issues/1075</p>
<p>In .NET, stack references do not support typed output values.</p>	<p>Use dictionaries or primitive types, or manually type cast when retrieving structured objects.</p>	<p>https://github.com/pulumi/pulumi/issues/12418</p> <p>https://github.com/pulumi/pulumi-dotnet/issues/58</p>
<p>Currently, Flexible FICs are not supported in Pulumi (it's a preview feature).</p>	<p>Use Azure CLI's <code>az rest</code> method to make direct REST API requests.</p> <p>This allows users to interact with Flexible FICs programmatically until official support.</p>	<p>https://learn.microsoft.com/en-us/entra/workload-id/workload-identities-flexible-federated-identity-credentials?tabs=terraformcloud#azure-cli-azure-powershell-and-terraform-providers</p>

Appendix B - Azure Issues & Workarounds

Description	Workaround	Further read
<p>Pulumi encounters issues when provisioning Azure AD directory roles because unassigned roles are not exposed via the Microsoft Graph API. This causes Pulumi to return null errors when trying to manage or reference these roles.</p> <p>The problem arises because Azure does not consider a directory role "activated" until it has been assigned to at least one user or service principal. Since Pulumi depends on the API to retrieve role information, any role that hasn't been explicitly assigned remains invisible to Pulumi, preventing it from being managed programmatically.</p>	<p>Manually assign the directory role to any user or service principal in the Azure portal before provisioning it with Pulumi.</p> <p>Once assigned, Azure marks the role as activated, allowing it to be detected via the API and managed by Pulumi.</p> <p>While this workaround ensures successful role provisioning, it defeats the purpose of automation and introduces manual overhead.</p>	<p>https://github.com/hashicorp/terraform-provider-azuread/issues/1526</p>